Python 源码解析读书笔记

—— 起于源码，不止于源码

Shell Xu 许智翔

Python 源码解析的意义和问题

- 了解系统底层，才能正确编码。
- 了解如何实现一个语言系统，对于模板系统设计 / 动态配置有指导意义。
- 针对 C 程序员，而非 Python 程序员。
- 讨论如何实现太多，为什么太少。

对象 - 类型的类型

```
obj int(10).ob_type -> PyInt_Type
PyInt_Type.ob_type -> PyType_Type
PyInt_Type.tp_base -> PyBaseObject_Type
PyBaseObject_Type.ob_type -> PyType_Type
PyType_Type.ob_type -> PyType_Type
```

更精确的参考源码解析 262 页图。

对象 - 小整数对象

```
#ifndef NSMALLNEGINTS
#define NSMALLNEGINTS           5
#endif

    if (-NSMALLNEGINTS <= ival && ival <
NSMALLPOSINTS) {
        v = small_ints[ival +
NSMALLNEGINTS];
        Py_INCREF(v);
    }
```

对象 - 大整数对象，空对象池，对象缓存

```
>>> a = 1000000
>>> b = 2000000
>>> id(a) == id(1000000)
False
>>> id(100000) == id(100000)
True
```

对象 - 字符串对象复用和缓存

```
>>> c = 'qazwsxedcrfvt'
>>> c += 'gbyhnujmikolp'
>>> a = 'qazwsxedcrfvtgbyhnujmikolp'
>>> id(a) == id(c)
False
>>> a = 'abc'
>>> b = 'def'
>>> c = 'abc'
>>> id(a) == id(c)
True
>>> b = 'abcde'
>>> id(b[1]) == id(c[1])
True
```

对象-free_list对象缓存的机制

每个类别有自己的free_list对象，用于缓存已经被销毁的对象。

目前尚不清楚GC是否会定时释放这部分内存，但是python在对象引用到0时是不释放对象的，而且多数情况下表现为内存泄漏。

而且多种对象的free_list不能互相通用，继承子类也不适用。

对象 -list 对象的行为

list 对象用一种 vector 等同的方法处理对象池。因此随机插入（尤其是头部插入）一个对象超长队列会引发大量的内存复制行为。

对象-dict对象的索引方案

dict对象的索引方案使用的是哈希表，而且是开放地址法的哈希表。当装载率达到一定规模后，会新申请一块内存，将有效数据复制过去。

最小的表空间为8个对象，当装载率超过2/3时，会扩大规模到当前active的4倍（超过50000个对象为2倍）。

目前为止，在对象被删除后，其表空间并不释放。因此曾经增长的非常大的dict对象，可以定期复制以回收空间。

对象-dict的用法注释

从序列中移除重复对象

```
dict.fromkeys(seqn).keys()
```

计算序列中元素出现次数

```
for e in seqn: d[e] = d.get(e,0) + 1
```

词典中移除大量元素

```
d = dict([(k, v) for k, v in d.items() if k != xxx])
```

词典中访问可能不存在的元素（当不存在的风险高于5%时）

```
o = d.get(k, default)
```

词典中访问可能不存在的元素（当不存在的风险低于5%时）

```
try: o = d[k]
except KeyError: o = default
```

函数 - 函数的性质 1

```
>>> def outer(o1, o2):
...     def inner(i1 = 10, i2 = []):
...             return i1+o1+o2
...     return inner
...
>>> a1 = outer(50, 30)
>>> a2 = outer(50, 30)
>>> a1.func_closure
(<cell at 0xb75454f4: int object at 0x8455ddc>, <cell at
0xb7545524: int object at 0x8455cec>)
>>> a2.func_closure
(<cell at 0xb754541c: int object at 0x8455ddc>, <cell at
0xb75453a4: int object at 0x8455cec>)
```

两次生成的函数对象拥有不同的闭包空间。

函数 - 函数的性质 2

```
>>> a1.func_defaults
(10, [])
>>> a2.func_defaults
(10, [])
>>> a1.func_defaults[1].append(10)
>>> a1.func_defaults
(10, [10])
>>> a2.func_defaults
(10, [])
```

也拥有不同的默认值空间。

```
>>> def default_test(d = []):
...     print d
...
>>> default_test.func_defaults
([],)
>>> default_test.func_defaults[0].append(10)
>>> default_test()
[10]
```

然而同一次生成的默认值空间是共享的，哪怕多次运行。

函数 - 参数传递 1

```
>>> def f(a,b,c,d): return a,b,c,d
...
>>> f(1,2,3,4)
(1, 2, 3, 4)
>>> f(1,2,a=3,b=4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() got multiple values for keyword argument 'a'
>>> f(1,2,c=3,d=4)
(1, 2, 3, 4)
```

参数分两种，位置参数和键值参数。具体如何传递是由调用时决定而非编译时。调用时参数必须先以位置参方式传递，再以键值参方式传递。一旦出现键值传递，再出现位置传递即出现编译时非法。调用时会先入栈所有参数，一个位置参占一个对象，一个键值参占两个对象（这是当然）。

函数 - 参数传递 2

解析的时候按照先位置后键值的方式赋值，先将所有位置参依次赋值给所有参数名。如果位置参有多，而没有扩展位置参来接收，则报错 TypeError: %s expected at %s %d arguments, got %d 。而后将所有键值参赋值给未赋值的参数，如果这个参数名已经赋值，则如上文，报错。如果键值参数有多，又没有扩展键值参来接受，也报错。
最后，如果有参数名尚未赋值，查看这些参数名是否有默认值。如果没有，报错。
另外，在字节码中访问本地（locals）命名空间的时候，是不通过命名空间查询的方式进行的。因为编译时可以明确一个名称是否在 locals 空间中，而不用理会代码段在名称空间中的位置结构。而一旦明确其在 locals 命名空间中，则可以直接堆栈访问位置，这样使得 locals 名称查询速度远高于普通名称空间。对于一个函数内频繁使用的符号，建议做一次赋值，将其引入 locals 命名空间。

函数 - 调用堆栈

python 的调用堆栈是通过 PyFrameObject 来实现的，每一次调用， python 会产生一个新的 PyFrameObject 加入到栈中。而每个 PyFrameObject 自带一个小数据区域，用于接收参数，处理局部变量。 python 字节码指令中的 LOAD_FAST ， STORE_FAST 就是操作的这个区域。

函数 - 层级闭包的实现 1

```
>>> def f1():
...     def f2(): return i
...     i = 10
...     return f2
...
>>> a = f1()
>>> a()
10
```

实现的还是不错的。通过计算当时名称 - 值的方法就无法获得 i 。

函数 - 层级闭包的实现 2

```
>>> def f1():
...     def f2():
...             return inet_aton
...     from socket import *
...     return f2
...
<stdin>:1: SyntaxWarning: import * only allowed at module
level
  File "<stdin>", line 4
SyntaxError: import * is not allowed in function 'f1'
because it is contains a nested function with free
variables
```

这主要是因为闭包的实现是通过函数编译时名称层状传递。例子 1 在编译时，f2 知道上层作用域中有一个名叫 i 的变量，于是 f2 的 freevars 属性就为 i 。而当 f1 操作 i 时，f2 保持了一个对结果的引用。当 f1 返回 f2 函数对象时，自身的 PyFrameObject 消失了没错，但是 f2 中对结果的引用还保存在了 func_closure 中。当 from socket import * 的时候，当前 locals 空间名称会发生变化，从而导致动态引入的名称无法在 f2 中生效。

对象和函数 -mro

算法，自身先入栈，而后按声明顺序继承每个父类的 mro，内部对象在最后。简单来说，深度优先，从左向右。

当类对象创建时，会将父类所有函数全部复制过来（很明显，应当是符号复制）。

对象和函数 -super 规则

```
>>> class A(object):
...     def f(self): print 'A'
...
>>> class B(object):
...     def f(self): print 'B'
...
>>> class C(A):
...     def f(self): print 'C'
...
>>> class D(C, B):
...     def f(self): super(D,
self).f()
...
>>> d = D()
>>> d.f()
C
>>> D.__base__
<class '__main__.C'>
>>> D.__bases__
(<class '__main__.C'>, <class
'__main__.B'>)
```

```
>>> class A(object):
...     def f(self): print 'A'
...
>>> class B(object):
...     def g(self): print 'B'
...
>>> class C(A, B):
...     def f(self): super(C,
self).g()
...
>>> c = C()
>>> c.f()
B
>>> C.__mro__
(<class '__main__.C'>, <class
'__main__.A'>, <class
'__main__.B'>, <type 'object'>)
```

super 的算法是跟随 mro 次序，寻找非本类第一个符合名称的函数，调用之。

对象和函数-construct

```
instance = cls.__new__(cls, *args, **kargs)
  cls.__init__(instance, *args, **kargs)
```

对象和函数-bound method

```
>>> class A(object):
...     def f(self): pass
...
>>> a = A()
>>> a.__class__.__dict__['f']
<function f at 0xb7595454>
>>> a.f
<bound method A.f of <__main__.A object at
0xb75a1e6c>>
>>> a.f.im_self
<__main__.A object at 0xb75a1e6c>
```

bound method是一个函数对象和一个实例对象的集合。

对象和函数-descriptor1

```
>>> class A(object):
...     def __get__(self, obj, cls): return 'A.__get__ %s
%s %s' % (self, obj, cls)
...
>>> class B(object):
...     v = A()
...
>>> b = B()
>>> b.v
"A.__get__ <__main__.A object at 0xb75a1cac> <__main__.B
object at 0xb75a1cec> <class '__main__.B'>"
```

某个 instance 的属性查找顺序为, obj.__dict__, class 属性（按照 mro 顺序）。如果有 descriptor 则先于 obj.__dict__ 。

对象和函数-descriptor1

于是，这解释了一个问题。我们定义函数的时候，定义的都是“类函数”，即函数是类的成员。为什么最终函数会变成实例的成员呢？为什么又在调用时会自动产生一个 self 呢？

实例在查找的时候，会先查找 class 属性中的 descriptor 。假定 class 有成员函数 f ，当使用 obj.f 时，首先命中这个函数对象，因为这个对象是一个 descriptor 。即使我们对实例赋值，也无法改变这一行为，即类成员函数无法在实例上重载。 descriptor 在取值时，会被调用 __get__ 方法，这一方法有 obj 参数。于是函数对象的默认 __get__ 返回了一个 bound method ，其中包含了 self 和函数对象自身。

这种行为在每次调用时都会发生，因此实例成员函数的性能比 unbound method 直接写对象要慢。

杂项 -GIL 的影响

很多人讨论 python 性能的时候都提到一个概念， GIL 。我在 python 源码中搜了一下，这个函数调用并不多，但是位置很要命。每个线程，生成的时候请求一下，退出的时候释放一下。在每次运行字节码前也会短暂的释放一下，让其他线程有获得运行的机会。说白了，除非程序显式的调用 release_lock 去释放资源，否则 python 是没有任何多线程能力的。这种机会并不很多，通常只发生在阻塞的时候。

而 python 原子化的粒度也比较清晰，就是每个字节码内部一定是原子的，字节码和字节码之间是非原子的。当我们操作 1.append 的时候，不用担心线程竞争导致数据结构损坏。但是如果我们操作 del 1[len(1)] 的时候，存在发生异常的概率。

杂项 - 对象缓存池

python 对小内存对象（碎片对象）提供了小内存对象缓存池。默认情况下， 256 字节以下的内存由小内存缓存池管理，以上的直接向系统申请，申请大小每 8 字节对齐。

对象缓存池的分配和收集技术采用了自由资源链表，在 2.5 之后，当某个尺度的资源不再需要时，会整体释放。

杂项 -python 的 GC 机制

python 的 GC 机制是基于引用计数的，因此当引用计数归零，对象一定会被释放（如果是碎片对象，内存不一定直接释放，可能归对象缓存池）。

python 的辅助垃圾收集算法是三色标记法和分代垃圾收集模型（generation），由于要跟踪所有的容器对象，因此容器对象上有跟踪链表。

杂项 - 字符编码处理方案

无论从何种来源，只要是字符串，并可能交给一个和当前代码并不紧密耦合的代码处理，就应当被转换为 unicode 。或者换一个更简洁的说法，应当使用 unicode 作为接口数据类型。

str 对象是很难猜测编码的，当离开了数据源代码后，再分析编码是个不靠谱的方案。